Ruby master - Feature #9999

Type Annotations (Static Type Checking)

06/30/2014 11:06 PM - DAddYE (Davide D'Agostino)

Status:	Feedback	
Priority:	Normal	
Assignee:		
Target version:		

Description

Hi all,

I know matz (Yukihiro Matsumoto) is interested in introducing **type annotations** in ruby. More here:
https://bugs.ruby-lang.org/issues/5583

I think it's time for ruby to get this.

Before working on a patch I would like to know:

1. Syntax of methods signatures
2. Syntax of variables guards (?)
3. Implementation

For point **1** I was thinking in some like:

```
def connect(r -> Stream, c -> Client) ->  Fiber
def connect(Stream r, Client c) -> Fiber
# quite sure this will make some reduce problems in the grammar
```

Before making a proposal consider: keyword arguments and default value collisions.

Then for point **2** I'm not sure if we want also check assignments but as before a syntax could be:

```
r: Client = something # will throw an exception if something is not kind of Client
```

Finally, **implementation**. Do we want some in python style and then leave the programmer/library for the implementation **or** (and I'm for this) we want MRI do that, if so how?

Cheers!
DD

p.s. Sorry if this issue was already discussed but I didn't find anything except the link posted.

History

#1 - 08/06/2014 04:11 AM - hsbt (Hiroshi SHIBATA)

- Status changed from Open to Assigned

#2 - 08/08/2014 01:04 PM - josh.cheek (Josh Cheek)

Regarding #1.

1. Syntax of methods signatures

Matz says "optional typing should honor duck typing", which this hasn't discussed.

There is already a decent amount of code in existence which does optionally describe type information via documentation. It can't currently be turned on (as in actually acted on by verifying that the types described match reality).

It seems reasonable to me, then, to build on what people are already doing, and using the YARD documentation types (http://yardoc.org/types), here is that code analyzed and gemified (https://github.com/pd/yard_types).

Next, we'd have to decide, do you put that information in a params list, or before the sig like in normal docs? I'm going to advocate before the sig, because I suspect there is a potential for type information to become too verbose to fit nicely in the params list, and placing it before the sig is, again,

closer to what people are already doing with docs.

If you like that idea, then the question becomes how to annotate it, is it additional Ruby syntax, or is it a new kind of magic comment? And again, I'll advocate a new kind of magic comment, because (1) it's closer to what people are already doing, (2) then this new feature becomes backwards compatible, because on older rubies it's just a normal comment (3) at this point, the implementation is so close to what documentation parsers are already doing anyway, that it might be possible for people's existing docs to suddenly become valid type verifications. Or, at the very least, they're incredibly close at this point.

Regarding #3

1. Implementation

I don't have a clue how leaving it up to a library would play out (ie, how well does this work for Python?) but I like that idea a lot. It would allow people to iterate on the capabilities provided in creative ways. Maybe it comes with a default Types::Off, but you can swap them out for Types::VerifyOnCall, allowing someone to swap them out for tests, or Types::CheckOnError, allowing them to turn off for prod, but then run type-checking if something blows up.

A proposition: Interfaces

I assume the purpose of this is to enable run-time type checking, which I'm not a big fan of, it seems like it would add a lot of overhead, and doesn't really feel like it can ever be correct -- in the sense that I probably can never adequately capture some types "takes an argument that implements :a, which takes 3 parameters and a block, a String, Fixnum, and something that responds to :to_s, but the String should have the additional method :status_code defined on its singleton class, which returns a Fixnum, and the block can take ...".

In other words, it seems like types need to capture the full path of everywhere that argument goes in code from this point forwards, and then also specify the param types and return types of every method we invoke. Unless we have an external definition that we are referencing back to, as is the case in your examples with Stream r. But that doesn't work for Duck typing, unless we also create interfaces.

Once we have interfaces, we could get useful things like:
1) Someone can specify this information much more tersely
2) Create test helpers like assert_implements for minitest, expect(obj).to implement SomeInterface for RSpec. Which would be super useful for anything that uses a plugin system with polymorphism. e.g. Capybara could turn Capybara::Driver::Base into such a thing (https://github.com/jnicklas/capybara/blob/304e2fbfe1e54702eb65f2a3feda1c7b9b99ff36/lib/capybara/driver/base.rb).
3) The potential exists to static type check everything. Well... at least in an incredibly large subset of code.

#3 - 09/21/2014 12:40 AM - roryokane (Rory O'Kane)

One design to consider is interfaces as implemented in the Go programming language, in which interfaces combine static typing and duck typing (https://en.wikipedia.org/wiki/Duck_typing#In_Go). Go accomplishes this by counting any object as conforming to an interface as long as it responds to the required methods. The object's type does not have to explicitly declare that it implements that interface, and it does not matter whether the interface was defined before or after the interface-implementing types were defined.

#4 - 09/30/2014 01:10 PM - shevegen (Robert A. Heiler)

I like the principal idea behind it.

For instance, today I wrote ruby code like this (yes, hate me for using set_ methods
but I like it visually because my brain works that way):

```
def set_be_verbose(i)
@be_verbose = i
end
```

@be_verbose can only be true or false, and must never be any value.

It can also only be modified through that accessor method above.

I thought it would be nice if I could somehow tell ruby to check automatically.

Of course I could check the input for FalseClass or TrueClass or something
but I wondered if it would not be better to also provide an additional
OPTIONAL way.

At any rate, I like the idea behind this - my main concern is only the
syntax.

The syntax proposal:

```
def connect(r -> Stream, c -> Client) ->  Fiber
```

I am sorry, that is awful. It would conflict with my ruby code (I do not
use ->) and the intent is not clear at all to me.

```
def connect(Stream r, Client c) -> Fiber
```

This is a bit better but it also uses -> and thus it really is not good
at all.

If we would not have to be backwards compatible we could use @@ hehe :)

At any rate, please consider the syntax! I picked ruby because it is
by far the most elegant language out there (actually, that is not
completely true, I picked ruby over python because of its philosophy
from matz' old interview from back then at: http://www.artima.com/intv/ruby.html
it really still is my favourite interview from matz hehe)

#5 - 12/17/2014 06:45 AM - pankajdoharey (Pankaj Doharey)

Proposed syntax above is this :

```
def connect(r -> Stream, c -> Client) ->  Fiber
def connect(Stream r, Client c) -> Fiber # quite sure this will make some reduce problems in the grammar
```

I think the proposed stabby Lambda syntax for typing is confusing, cause collisions and not very cool.
Possibly we could use something like the following.

```
def connect(r |> Stream, c |> Client) |>  Fiber
def connect(Stream r, Client c) |> Fiber # Will not create any reduce problems in the grammar
```

#6 - 12/17/2014 07:52 AM - gogotanaka (Kazuki Tanaka)

I'm not sure how type annotations, gradual type or something like that with Ruby is, but it worth thinking.

What I can say here now is not only class but also method can be annotated and typed attr_accessor is also worth thinking.

```
def sum(x, y)
  x.to_i + y
end
```

then, x can be annotated as "something responding to :to_i" or "something responding to some methods which respond to +"

Typed attr_accessor

```
attr_accessor name: String # assert :name String(or NilClass ..?)
```

Just idea : (

#7 - 12/17/2014 11:56 AM - recursive-madman (Recursive Madman)

How about

```
def connect<Fiber>(r<Stream>, c<Client>)
```

?

#8 - 03/22/2015 05:06 AM - mrkaspa (Michel Perez)

If you start to add types to ruby for me it will end looking similar to scala

def connect(r: Stream, c: Client): Fiber

#9 - 03/22/2015 10:35 AM - michalmuskala (Michał Muskała)

I think it would be quite interesting to see how other dynamic languages deal with types. I'm not sure adding runtime checks is the way to go - it adds overhead and complicates the runtime, and in my mind static typing has a purpose of speeding things up, and reducing method lookup overhead. Or at least leaving the thinks at the speed they were, but extending program's safety.

I think a good way to introduce type checks is through a separate process that will analyse the code. One example of such a system is Erlang's dialyzer.
It works using a mechanism they call "Success typing". At the beginning the system assumes every function accepts all types and can return any type. Later through analysis of the code it learns how you use the functions and warns you if it ever finds some contradictions (you can read more about it here: http://learnyousomeerlang.com/dialyzer).
It's purpose is not to guarantee 100% type safety, but to catch majority of the problems.
There's also a syntax for providing dialyzer with hints about function signature. In Ruby's case using existing YARD types or producing some additional simple syntax could be a way to go, for example:

```
#spec (Integer, Integer) :: Integer
```

```
def add(x, y)
  x + y
end
```

This syntax has also the great advantage of being backwards compatible (as for older rubies it's just a comment).

#10 - 05/03/2016 09:33 AM - coyote (Alexey Babich)

- Subject changed from Type Annotations to Type Annotations (Static Type Checking)

For usage level:
Static type checking looks most useful as syntax pre-compilation feature to speed-up code execution and code/syntax verification-before-execution improvements

For syntax:
Current syntax (e.g. 2.3*) shows that the most compatible syntax is

```
def ReturnType method(Type value: default, ...)
```

Also, we can have some simplified syntax for known defaults, e.g.

```
def method(count: 0!)
```

where "exclamation mark" means that type of default value is for Static type definition

```
>> 0.class
=> Fixnum
```

like equivalent to

```
def method(Fixnum count: 0)
```

For type definition:
It looks natural to stick with existing class definitions, not to redefine understanding of "type"
ruby should consider which classes can be optimised and how internally with some clear documentation about this

For value of the feature
It will be really nice to see some *experimental* implementations in 2.5 or even in 2.4 depending on priorities of core team to make it possible to try, benchmark and improve far before ruby 3

#11 - 05/03/2016 11:26 AM - nobu (Nobuyoshi Nakada)

Alexey Babich wrote:

> `def ReturnType method(Type value: default, ...)`

How to know if it doesn't define a method ReturnType?

#12 - 05/03/2016 12:36 PM - coyote (Alexey Babich)

Nobuyoshi Nakada wrote:

> Alexey Babich wrote:
>
> > `def ReturnType method(Type value: default, ...)`
>
> How to know if it doesn't define a method ReturnType?

As I know, currently syntax like

```
def B a(...); end
```

is not possible and raises smth like

```
SyntaxError: (irb):11: syntax error, unexpected '(', expecting ';' or '\n'
```

The same time

```
def B a; end
```

is ok, so my suggestion is not great for return type

The same time it looks weird if we can define

```
Class var: value
```

where value can be virtually kind of another Class, incl. nil (NilClass) value

Do some *whitelisting* for some classes like NilClass make the idea overcomplicated
it makes reasonable to do only simplified syntax

```
def method(var: value!)
```

the same time definition for method like

```
def method: value
```

makes no sense in Ruby unless it is reasonable to introduce default return value for empty return statement

Also, I think syntax question is not the most important here

Thanks!

#13 - 05/19/2016 12:25 AM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Rejected

We are not going to add any kind of type annotation to Ruby.
But as part of Ruby3x3 attempt, we are trying to add type inference (both static and dynamic).

Matz.

#14 - 05/20/2016 02:19 AM - ml@n-ary.org (Rob Blanco)

Yukihiro Matsumoto wrote:

> We are not going to add any kind of type annotation to Ruby.
> But as part of Ruby3x3 attempt, we are trying to add type inference (both static and dynamic).

There was a GSoC proposal accepted to add gradual typing to MRI, as written a while ago in the mailing list. It seems relevant to mention here, although it may be the case that MRI isn't presently a target where this is desired. (There is little doubt in my mind that type inference has an important role to play either way. However, I'm not so sure how far it can go without at least some type annotations.)

#15 - 05/20/2016 10:01 AM - naruse (Yui NARUSE)

- Status changed from Rejected to Feedback

- Assignee deleted (matz (Yukihiro Matsumoto))

As matz says Ruby itself doesn't have a plan to include type annotations as its language syntax.
Therefore people who want to add type annotations must design them as outside of syntax, for example rdoc, comment, or something.

(I personally believe Ruby should have a way to annotate types or something)

#16 - 06/23/2016 03:53 PM - jrochkind (jonathan rochkind)

Existing ruby has interesting bits of a kind of formal duck-typing for some core library classes, like String#to_s and #to_str. There are a lot of methods that want 'a String', and will silently accept anything that can be made one with to_str (or in some cases to_s? not sure), otherwise raise a TypeError.

It would be interesting to incorporate this into a more formal type annotation system. If an argument is annotated as being String, maybe it will happily accept anything with a to_str, silently calling it?

Maybe extensible, so any class can define it's own "as if" method, String.type_conversion_method = :to_str, and if an argument type is annotated as being SomeClass, then any argument will have SomeClass.type_conversion_method called on it if possible.

Not sure of all the details, there is some messiness in the existing patterns (when/whether to_s vs to_str is already sometimes confused; having to add a conversion method to every class that can be converted isn't quite right). But something along these lines might be a good idea to actually be consistent with existing Ruby conventions instead of just bolting on a completely new system that has nothing to do with the existing conventions, as well as having an optional typing system still have some notion of duck typing/typing to interface, which are just good OO design principles in addition to being part of ruby community norms.

#17 - 08/07/2016 08:49 PM - michaelmior (Michael Mior)

Might be worth looking at RDL for some inspiration.

#18 - 04/03/2017 05:52 PM - burlesona (Andrew Burleson)

RDL is interesting, I wonder what the runtime overhead is like?

Another source of inspiration could be Facebook's Flow (https://flow.org/en/docs/getting-started/) for JS. In that case it's build-time type checking, mostly inferred, with optional annotations. While it would be nice to avoid mandatory annotations, and I see Matz saying they won't exist at all, being able to add optional annotations would be nice. /shrug